Security Auditing Report

Frame Electron Application

Prepared for: Jordan Muir, Frame Labs
Prepared by: Mykhailo Baraniak, Luca Carettoni
Date: 03/14/2022

Table of Contents

Revision History

Version	Date	Description	Author
1	03/11/2022	First release of the final report	Mykhailo Baraniak
2	03/14/2022	Document Peer Review	Luca Carettoni

Contacts

Company	Name	Email
Frame Labs, Inc.	Jordan Muir	jordan@frame.sh
Doyensec, LLC.	Luca Carettoni	luca@doyensec.com

Executive Summary

Overview

Frame Labs, Inc. engaged Doyensec to perform a security assessment of the Frame Electron application. The project commenced on 02/28/2022 and ended on 03/11/2022 requiring one security researcher. The project resulted in 7 (seven) findings of which 1 (one) was rated as high severity.

The project consisted of a manual Electron application security assessment.

Testing was conducted remotely from Doyensec EMEA and US offices.

Scope

Through meetings with Frame Labs, Inc. the scope of the project was clearly defined. We list the agreed upon assets below:

- Frame Electron application v0.5
  - macOS, Windows, Linux versions
- Integration of the Frame Electron and eth-provider client

The testing took place in the production environment using the latest version of the software at the time of testing.

Specifically, this activity was performed on the following releases:

- https://github.com/floating/frame-audit
  - 6219adfda47d06d2d755e4e4db67d28cd7e8496b
- https://github.com/floating/eth-provider
  - 804e8b7b3a5fc6dc2ca0699949ff99bf6ba68e2a

Scoping Restrictions

During the engagement, Doyensec did not encounter any difficulties while testing the desktop platform.

Frame engineers were very responsive throughout the entire engagement.

Findings Summary

Doyensec researchers discovered and reported 7 (seven) vulnerabilities in Frame's Electron application. While most of the issues are departure from best practices and low-severity flaws, Doyensec identified 1 (one) issue rated as high severity.

It is important to reiterate that this report represents a snapshot of the security posture of the environment at a point in time.

Major findings include the ability to force signing requests from untrusted origins and incorrect file permissions in use for configuration files.

Overall, the security posture of the Frame Electron application was found to be in line with industry's best practices.

At the design level, Doyensec found the system to be well architected with the exclusion of the following aspect:

- The notification message displayed during signing requests allows the use of special characters. Additionally, it lacks a visual representation (e.g. scrollbar) of the real message size. Critical security notifications (such as signature requests) must be concise yet clear to the user. The overall UX design should reduce the risk of phishing and UI redressing attacks.

Recommendations

The following recommendations are proposed based on studying Frame's security posture and vulnerabilities discovered during this engagement.

Short-term improvements

- Work on mitigating the discovered vulnerabilities. You can use **Appendix B** - Remediation Checklist to make sure that you have covered all areas

- Improve the platform's resilience against attack by implementing all defense in depth mechanisms specified within our *Low* and *Informational* findings

Long-term improvements

- Implement all defense in depth mechanisms specified within **Appendix C - Hardening**

- Increase your confidence in all third-party NPMs by:

    A. Locking all dependencies on specific versions within private forks
    B. Creating and maintaining a priority list based on project maturity, public scrutiny, functional vs security relevance
    C. Perform extensive gray-box testing on all NPMs (in order)
    D. Update dependencies whenever needed for new features or bug fixes

Methodology

Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key for standing against threats. Thus, we recommend a *graybox* approach combining dynamic fault injection with an in-depth study of source code to maximize the ROI on bug hunting.

During this assessment, we have employed standard testing methodologies (e.g. OWASP Testing guide recommendations) as well as custom checklists to ensure full coverage of both code and vulnerabilities classes.

Setup Phase

Frame provided access to the online environment, source code repository, and binaries for the wallet Electron-based application.

Doyensec had to instrument the desktop application to allow debugging and to facilitate testing. This was achieved with the use of `proxychains` and other tools.

Tooling

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve efficiency and efficacy of our effort.

During this engagement, we used the following tools:

- Burp Suite
- Sublime Text
- Remix IDE
- IPFS CLI
- Chromium Developers Tool
- Electronegativity
- Curl, netcat and other Linux utilities

Web Application and API Techniques

Web assessments are centered around the data sent between clients and servers. In this realm, the principle audit tool is the Burp Suite, however, we also use a large set of custom scripts and extensions to perform specific audit tasks. We focus on authorization, authentication, integrity, and trust. We study how data is interpreted, parsed, stored, and relayed between producers and consumers.

We subvert the client with malicious data through reflected and DOM based Cross Site Scripting and by breaking assumptions in trust. We test the server endpoints for injection style flaws including, but not limited to, SQL, template, XML, and command injection flaws. We look at each request and response pair for potential Cross Site Request Forgery and race conditions. We study the application for subtle logic issues, whether they are authorization bypasses or insecure object references. Session storage and retrieval is scrutinized and user separation is thoroughly tested.

Web security is not limited to popular bug titles. Doyensec researchers understand the goals and needs of the application to find ways of breaking the assumed control flow.

Electron Apps Testing

Doyensec has been the first security company to publish a comprehensive security overview of the Electron framework during BlackHat USA 2017. Since then, we have reported dozens of vulnerabilities in the framework itself and popular Electron-based applications.

Thanks to our research efforts, we have extensive experience in analyzing desktop runtime environments based on web technologies. During our testing effort, we will review security mechanisms that ensure isolation between sites, facilitate web security protections, and prevent untrusted remote content to compromise the security of the host. We write custom tools to map out control flow and study an application's behaviour and internals. Mapping out an attack surface, whether local or remote, is paramount to a successful engagement. Doyensec studies the application's ecosystem, looking for potential downfalls and misconceptions.

Static analysis and instrumentation are important parts of the testing process we use to test an application's response to untrusted data. Doyensec combines manual security testing with a mature Electron.js security testing tool (https://github.com/doyensec/electronegativity) which will be customized to meet the needs of the target.

We take apart the application looking for privacy leaks and secrets. Storage, transmission, and protection of user information is critical.

Project Findings

The table below lists the findings with their associated ID and severity in the order of discovery. The severity ranking and vulnerability classes are defined in **Appendix A** at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing.

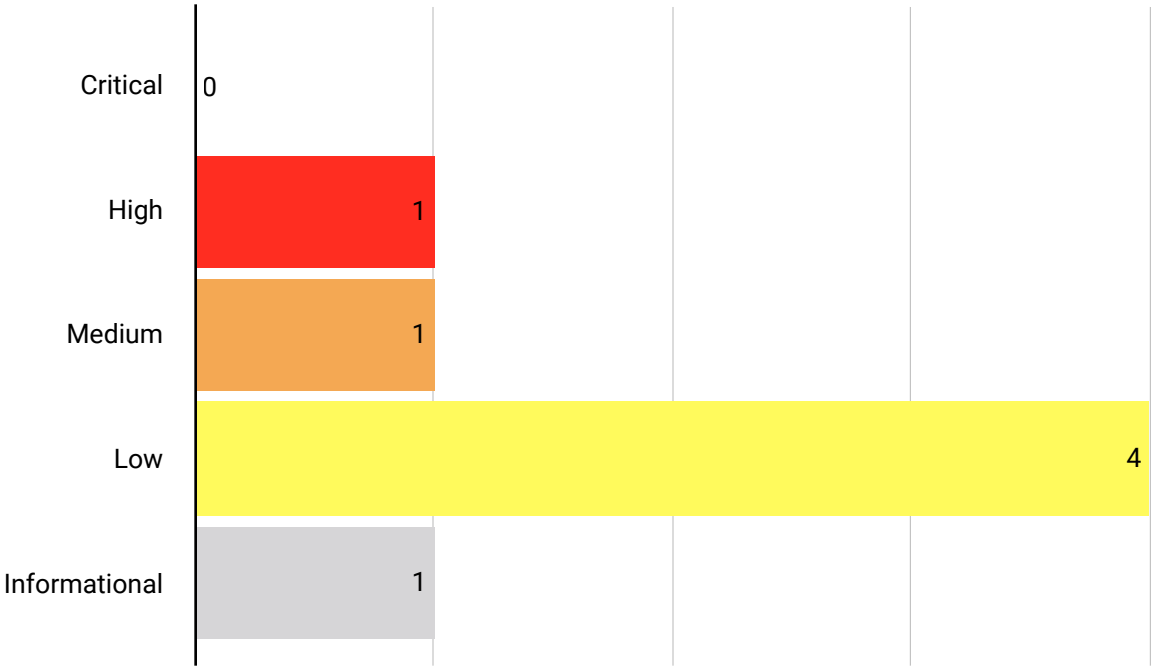
This table is organized by time of discovery. The issues at the top were found first while those at the bottom were found last. Presenting the table in this fashion has a number of benefits. It inherently shows the path our auditing took through the target and may also reveal how easy or difficult it was to discover certain findings. As a security engagement progresses, the researchers will gain a deeper understanding of a target which is also shown in this table.

Findings Recap Table

ID	Title	Vulnerability Class	Severity	Status
FRA-Q122-1	Outdated Electron Version	Components With Known Vulnerabilities	Informational	Open
FRA-Q122-2	Hard-Coded Credentials	Information Exposure	Low	Open
FRA-Q122-3	Settings and Signers Incorrect File Permissions	Security Misconfiguration	Medium	Open
FRA-Q122-4	Ability to Force Signing Requests From Untrusted Origins	Security Misconfiguration	High	Open
FRA-Q122-5	shell.openExternal validHost Bypass	Security Misconfiguration	Low	Open
FRA-Q122-6	Any Browser Extension Can Bypass The Origin Check	Security Misconfiguration	Low	Open
FRA-Q122-7	Possibility To Forge Signing Notification Message	Insecure Design	Low	Open

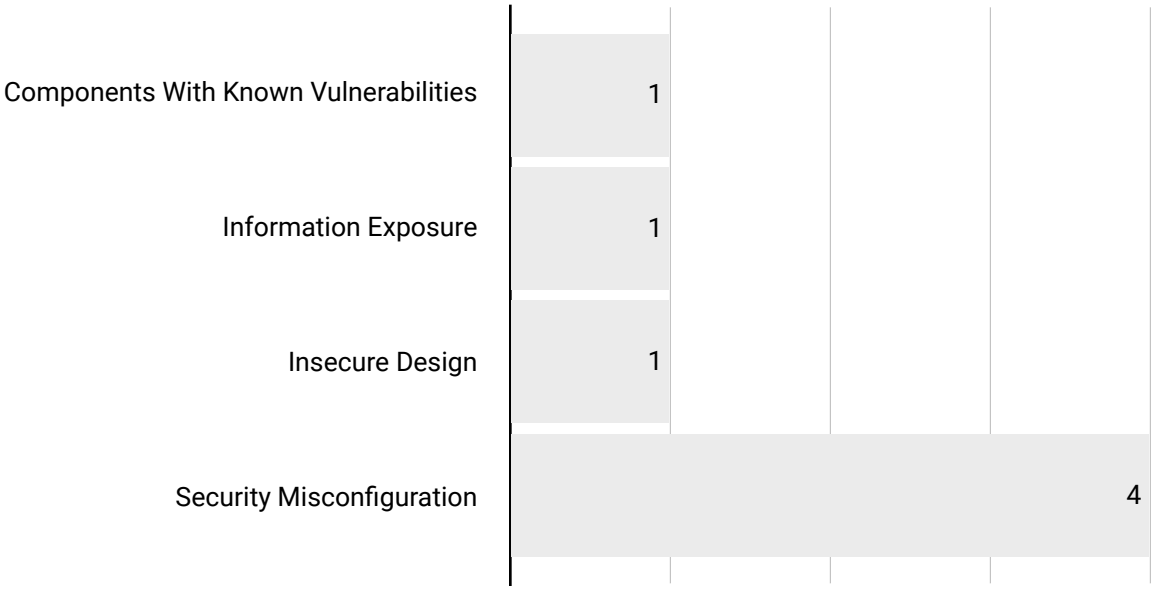
Findings per Severity

The table below provides a summary of the findings per severity.



Findings per Type

The table below provides a summary of the findings per vulnerability class.



FRA-Q122-1 - Outdated Electron Version	
Severity	**Informational**
Vulnerability Class	Components With Known Vulnerabilities
Component	frame-audit/package.json:182
Status	Open

Description

While reviewing the application's dependencies, Doyensec discovered that the current version of ElectronJS in use is affected by known vulnerabilities.

Dependecy	Installed Version	Vulnerability	Severity
Electron	15.3.2	CVE-2022-0609, CVE-2022-0610, CVE-2022-0607, CVE-2022-0608, CVE-2022-0606, CVE-2021-38012, CVE-2021-38017, CVE-2021-38019, CVE-2021-4066, CVE-2021-4100	High

Impact

The aforementioned Electron version lacks back-ported fixes for multiple security issues that may be exploited to attack the Frame Electron application. The installed package is vulnerable to: Use After Free, Inappropriate implementation in Gamepad API, Integer Overflow, Type Confusion vulnerabilities and others.

Complexity

High. While this dependency is affected by known vulnerabilities, the exploitation through Frame is considered of high complexity. Considering impact and complexity, the severity of this issue has been marked as "Informational".

Remediation

Upgrade the aforementioned dependencies to the latest version or at least to a version which is not vulnerable to the vulnerabilities above.

Resources

- "Electron Stable Releases
  https://www.electronjs.org/releases/stable?version=15

- OWASP, Top 10-2017 A9-Using Components with Known Vulnerabilities
  https://www.owasp.org/index.php/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities

FRA-Q122-2 - Hard-Coded Credentials	
Severity	**Low**
Vulnerability Class	Information Exposure
Component	frame-audit/dash/store/index.js:40, frame-audit/flow/store/index.js:35, frame-audit/main/accounts/index.ts:214, frame-audit/main/contracts/index.ts:48
Status	Open

Description

While auditing the source code, Doyensec identified the use of hard-coded credentials[1].

Hard-coded credentials are plain text usernames, passwords, or secrets stored in the source code, or within the code repository.

Reproduction Steps

This is a source code finding. The following resources are affected:

frame-audit/dash/store/index.js:40
```
//fetch('https://api.etherscan.io/api?
module=stats&action=ethprice&apikey=KU5RZ9156Q51F592A93RUKHW1...'
```

frame-audit/flow/store/index.js:35
```
//fetch('https://api.etherscan.io/api?
module=stats&action=ethprice&apikey=KU5RZ9156Q51F592A93RUKHW1...'
```

frame-audit/main/accounts/index.ts:214
```
fetch('https://api.etherscan.io/api?
module=stats&action=ethprice&apikey=KU5RZ9156Q51F592A93RUKHW1...'
```

frame-audit/main/contracts/index.ts:48
```
const res = await fetch(`https://api.etherscan.io/api?
module=contract&action=getsourcecode&address=${contractAddress}
&apikey=3SYU5MW5QK8RPCJ...`)
```

Impact

Medium. Since Frame's source code is public, an attacker can overuse 3'd party API keys and completely use available quota, rendering Etherscan unavailable for the Frame Electron application. The maximum

1 https://cwe.mitre.org/data/definitions/798.html

tier for the Etherscan API key is 30 requests per second only.

Complexity

Low. An attacker can obtain source code from the public Git repository and write a simple script to perform DoS attack on the application.

Remediation

Don't store API keys within the code. Always store your credentials on the server side. Fetch the API results from there and then pass them to the client application. In the OSS codebase, use configuration settings instead of hard-coded keys and secrets.

Resources

- MITRE, “CWE-798: Use of Hard-coded Credentials”
  https://cwe.mitre.org/data/definitions/798.html

FRA-Q122-3 - Settings and Signers Incorrect File Permissions	
Severity	**Medium**
Vulnerability Class	Security Misconfiguration
Component	config.json, signers/*.json
Status	Open

Description

When a resource is given a permissions setting that provides access to a wider range of actors than required, it could lead to the exposure of sensitive information, or the modification of that resource by unintended parties. This is especially dangerous when the resource is related to program configuration, execution, or sensitive user data.

While auditing the installed application, Doyensec identified that the Frame Electron application specifies permissions for security-critical resources in a way that allows read/write by unintended actors. Please refer to the reproduction steps below.

Reproduction Steps

1. Instrument the Frame Electron application to store user data in the predefined location. Modify the `frame-audit/main/index.js` file:

```
app.setPath ('userData', "/tmp/user_data_folder");
```

2. Build the application and add a new account
3. Verify permissions of the `config.json` and `signers/*` files:

```
-rw-rw-rw- 1 myuser staff 30317 Mar 5 20:15 config.json

-rw-r--r-- 1 myuser staff 4974  Mar 5 19:54
6161346338623333231646363323131623536303631323338363231343661 66.json
```

Impact

High. A low privileged local attacker can modify application configurations. For example, change permissions for the external origins. It is also possible to read encrypted private keys.

Complexity

Medium. A local attacker can trivially change the configuration file with any text editor. This can affected installations on shared computers and within organization workstations.

Remediation

Change permissions for the sensitive files listed above. For all configuration files, executables, and libraries, make sure that they are only readable and writable by the current user only.

```
const fs = require('fs');
fs.chmodSync('config.json', 0o600);
```

Resources

- MITRE, “CWE-732: Incorrect Permission Assignment for Critical Resource” https://cwe.mitre.org/data/definitions/798.html

FRA-Q122-4 - Ability to Force Signing Requests From Untrusted Origins	
Severity	**High**
Vulnerability Class	Security Misconfiguration
Component	frame-audit/main/provider/index.ts:946
Status	Open

Description

The Frame Electron application verifies that state changing requests could be sent from trusted origins (domains approved by the user and reflected in the permission bar) only.

Nevertheless, Doyensec discovered that it is possible to send a signing request from untrusted (hence not connected) origins. The root cause of this issue lays in the way `signTypedDataRequest` is handled.

```
// frame-audit/main/provider/index.ts:946
      const signTypedDataMatcher = /eth_signTypedData_?(v[134]|$)/
      const signTypedDataRequest = method.match(signTypedDataMatcher)

      if (signTypedDataRequest) {
        const version = (signTypedDataRequest[1] || 'v1').toUpperCase() as Version
        return this.signTypedData(payload, version, res)
        }

// frame-audit/main/api/http.js:46
       if (protectedMethods.indexOf(payload.method) > -1 && !(await trusted(origin)))
       {
              let error = { message: 'Permission denied, approve ' + origin + ' in
     Frame to continue', code: 4001 }

// frame-audit/main/api/protectedMethods.ts
      export default [
      ...
        'eth_sign',
        'eth_signTypedData',
        'eth_signTypedData_v1',
        'eth_signTypedData_v3',
        'eth_signTypedData_v4',
      ...
```

`String.match` method will return an array whose contents depend on the presence or absence of the global (g) flag, or null if no matches are found. Code execution is shown below:

```
      let method = "eth_signTypedData_v1.1"
      const signTypedDataMatcher = /eth_signTypedData_?(v[134]|$)/
      let signTypedDataRequest = method.match(signTypedDataMatcher)

      > signTypedDataRequest
```

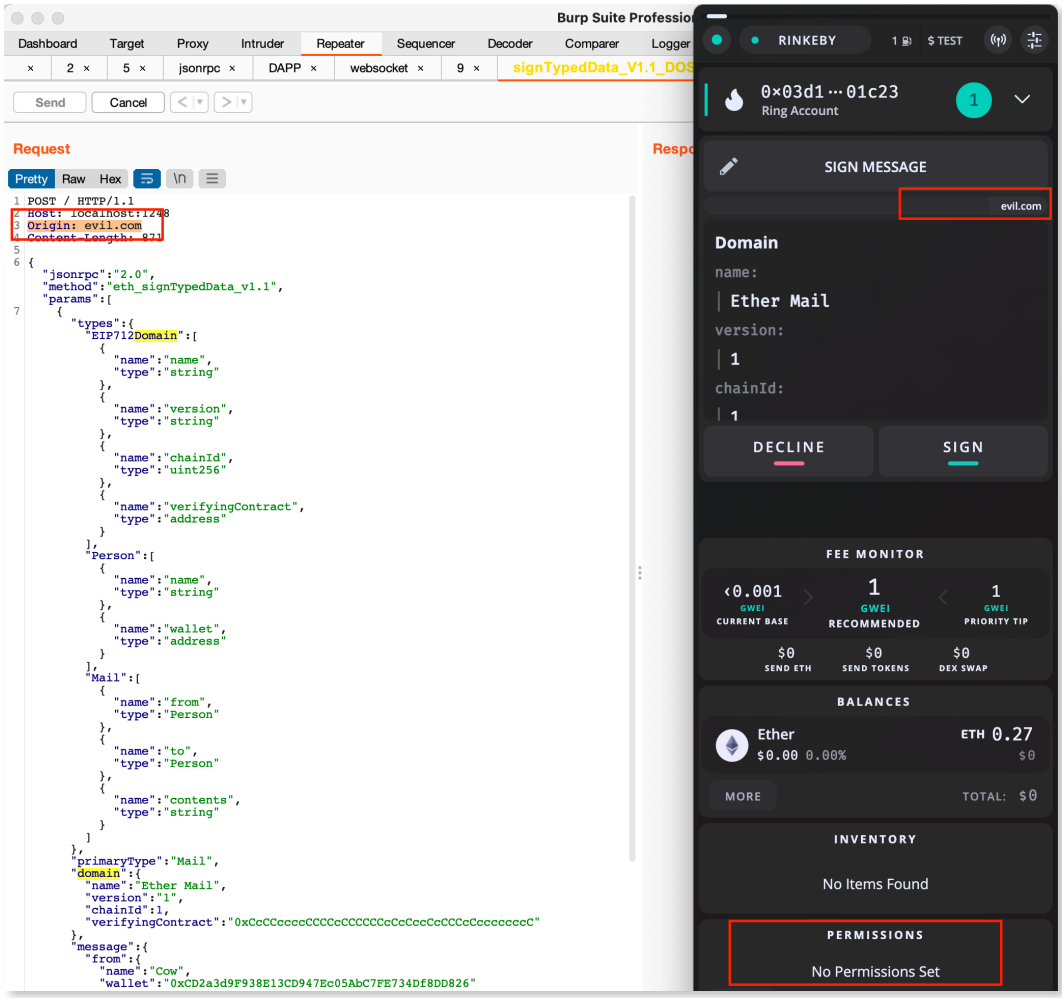
```
[
  'eth_signTypedData_v1',
  'v1',
  index: 0,
  input: 'eth_signTypedData_v1.1',
  groups: undefined
]

> const version = (signTypedDataRequest[1] || 'v1').toUpperCase()
> version
'V1'
```

Reproduction Steps

Follow the steps below to reproduce this issue:

1. The victim user opens a malicious web page in any browser of choice.
2. An attacker's script sends POST request with the `eth_signTypedData_v1.1` method to the local JSON-RPC endpoint.



Please note that the user has not connected to any domain, but still got a signing request.

Impact

The attacker can send a signing request from unconnected external domains. To sign the message, the victim still needs to explicitly click the "Sign" button.

It is also possible to abuse this issue in order the render the application unavailable to users. The attacker can send a signing message with invalid structure (e.g. empty json string). After that, the user can only kill the Frame application from the command line. Frame's UI is completely empty and non responsive.

Complexity

Low. Only basic web skills are required to send a malicious request. One limiting factor of this issue is the fact that the payload must contain a valid user's account.

Considering impact and complexity, the severity of this issue has been marked as “High”.

Remediation

Strictly validate JSON-RPC method name.

```
if (method === eth_signTypedData || method === eth_signTypedData_v1 || method === eth_signTypedData_v3 || method === eth_signTypedData_v4) {
    ...
```

Additionally, add to the list of protected methods the dangerous `eth_sendRawTransaction` method.

Resources

- “Ethereum JSON-RPC API”
  https://eth.wiki/json-rpc/API

FRA-Q122-5 - shell.openExternal validHost Bypass	
Severity	**Low**
Vulnerability Class	Security Misconfiguration
Component	frame-audit/main/index.js:142
Status	Open

Description

While auditing the source code, Doyensec discovered that the `shell.openExternal` function is not correctly protected by the `validHost` check.

The verification step involves the use of `startsWith` and multiple entries in the `externalWhitelist` list lack a trailing slash character `(/)`. As a result, an attacker can register a domain that would bypass this check (e.g. https://frame.sh.doyensec.com).

Reproduction Steps

This is a source code finding. Open `frame-audit/main/index.js:142` with a text editor of your choice and verify the application logic

```
ipcMain.on('tray:openExternal', (e, url) => {
  const validHost = externalWhitelist.some(entry => url.startsWith(entry))
  if (validHost) shell.openExternal(url)
})

const externalWhitelist = [
  'https://frame.sh',
  'https://chrome.google.com/webstore/detail/frame-alpha/
ldcoohedfbjoobcadoglnnmmfbdlmmhf',
  'https://addons.mozilla.org/en-US/firefox/addon/frame-extension',
  'https://github.com/floating/frame/issues/new',
  'https://github.com/floating/frame/blob/master/LICENSE',
  'https://github.com/floating/frame/blob/0.5/LICENSE',
  'https://aragon.org',
  'https://mainnet.aragon.org',
  'https://rinkeby.aragon.org',
  'https://shop.ledger.com/pages/ledger-nano-x?r=1fb484cde64f',
  'https://shop.trezor.io/?offer_id=10&aff_id=3270',
  'https://discord.gg/UH7NGqY',
  'https://frame.canny.io',
  'https://feedback.frame.sh',
  'https://wiki.trezor.io/Trezor_Bridge',
  'https://opensea.io'
]
```

Impact

High. An attacker can bypass the `validHost` check and pass malicious domain to the `shell.openExternal`. For example, `https://frame.sh.doyensec.com` will successfully pass the `validHost` verification.

Complexity

High. The biggest barrier in exploiting this issue is finding another vulnerability to send a RPC message to the main Electron process with a `'tray:openExternal'` message.

Remediation

Add trailing slashes to the external whitelisted domains.

```
const externalWhitelist = [
        'https://frame.sh/',
        'https://chrome.google.com/webstore/detail/frame-alpha/
      ldcoohedfbjoobcadoglnnmmfbdlmmhf',
        'https://addons.mozilla.org/en-US/firefox/addon/frame-extension',
        'https://github.com/floating/frame/issues/new',
        'https://github.com/floating/frame/blob/master/LICENSE',
        'https://github.com/floating/frame/blob/0.5/LICENSE',
        'https://aragon.org/',
        'https://mainnet.aragon.org/',
        'https://rinkeby.aragon.org/',
        'https://shop.ledger.com/pages/ledger-nano-x?r=1fb484cde64f',
        'https://shop.trezor.io/?offer_id=10&aff_id=3270',
        'https://discord.gg/UH7NGqY',
        'https://frame.canny.io/',
        'https://feedback.frame.sh/',
        'https://wiki.trezor.io/Trezor_Bridge',
        'https://opensea.io/'
      ]
```

Additionally, we would strongly recommend to enforce a strict string matching.

Resources

- “ElectronJS Shell”
  https://www.electronjs.org/docs/latest/api/shell

- “RFC 3986 - URI Generic Syntax”
  https://datatracker.ietf.org/doc/html/rfc3986

FRA-Q122-6 - Any Browser Extension Can Bypass The Origin Check	
Severity	**Low**
Vulnerability Class	Security Misconfiguration
Component	frame-audit/main/api/isFrameExtension.js frame-audit/main/api/ws.js:29
Status	Open

Description

While auditing the source code, Doyensec discovered that the `isFrameExtension` function is just verifying whether the websocket message is received from a browser extension. Any browser extension is able to successfully pass this check.

Reproduction Steps

Open `frame-audit/main/api/ws.js` with a text editor of your choice and verify application logic

```
socket.isFrameExtension = isFrameExtension(req)
...
socket.on('message', async data => {
    let origin = socket.origin
    const payload = validPayload(data)
    if (!payload) return console.warn('Invalid Payload', data)
    if (socket.isFrameExtension) { // Request from extension, swap origin
      if (payload.__frameOrigin) {
        origin = payload.__frameOrigin
        delete payload.__frameOrigin
      } else {
        origin = 'frame-extension'
      }
    }
```

Open `frame-audit/main/api/isFrameExtension.js`

```
const queryString = require('query-string')
module.exports = req => {
  const origin = req.headers.origin
  if (!origin) return false
  const query = queryString.parse(req.url.replace('/', ''))
  if (origin.indexOf('chrome-extension://') > -1 || origin.indexOf('moz-
extension://') > -1) return query.identity === 'frame-extension'
  return false
}
```

Impact

High. Any installed extension can masquerade as a valid Frame extension and abuse its permissions.

Complexity

High. The biggest barrier is manipulating a victim to install browser extension with a malicious code. Considering the impact and complexity, the severity of this issue has been marked as “Low” only.

Remediation

Ensure that the `isFrameExtension` function allows whitelisted extensions only.

Every app and extension in the Chrome Web Store has its own unique identification (ID) that doesn’t change across versions. The application should specifically check for Frame's extension IDs.

Resources

- “Set Chrome app and extension policies” https://support.google.com/chrome/a/answer/7532015?hl=en

FRA-Q122-7 - Possibility To Forge Signing Notification Message	
Severity	**Low**
Vulnerability Class	Insecure Design
Component	Message Signing / UX
Status	Open

Description

The Frame Electron application creates sign notifications with user supplied data.

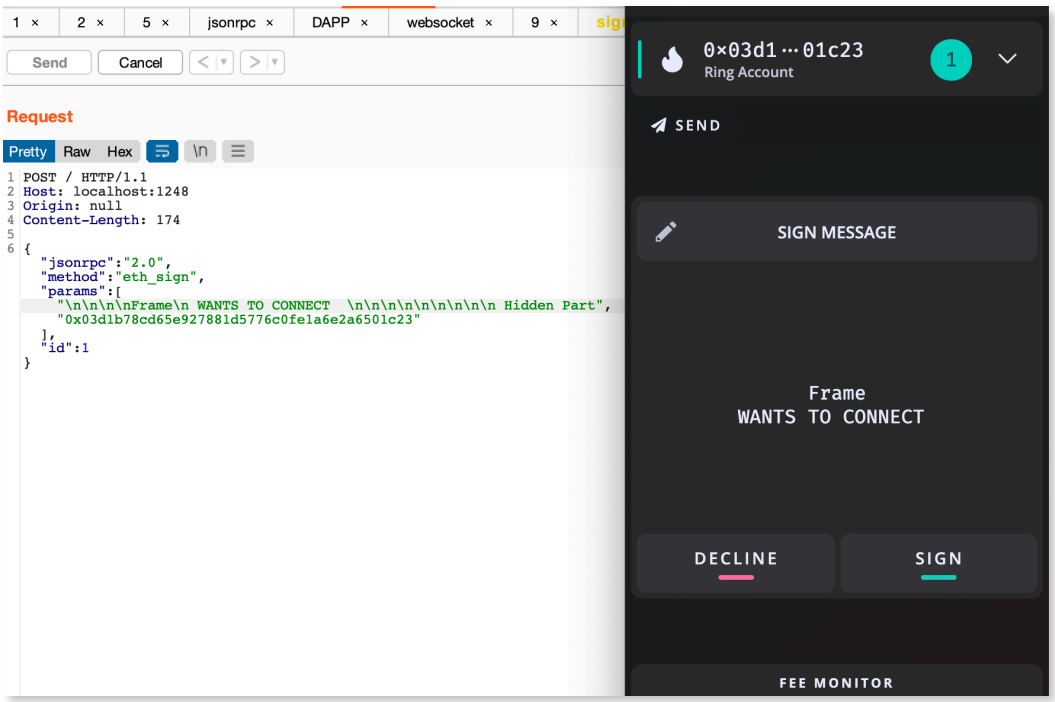
This data is shown without screen boundaries and special characters checks. As a result, an attacker is able to forge signing message notifications similar to the connection notifications, hiding malicious content with new line characters `\n`.

UI-redressing attacks are particularly dangerous in this context since such notifications might easily persuade a victim to perform unwanted interactions.

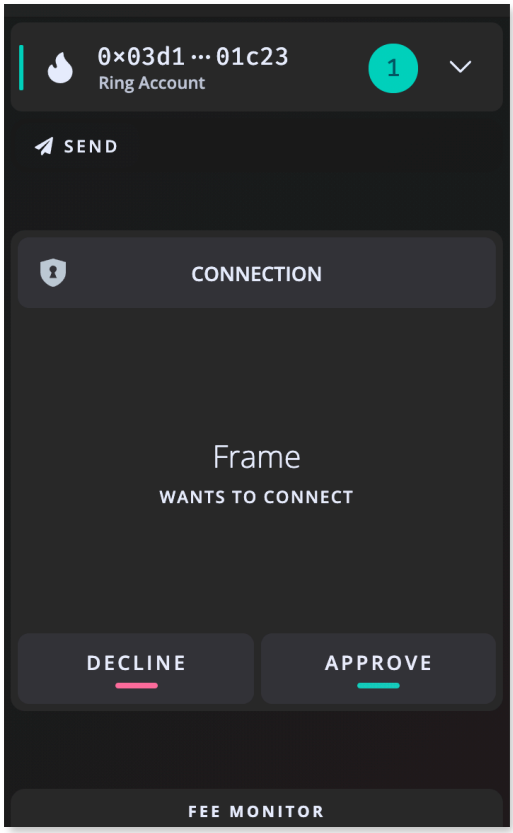
Reproduction Steps

As an example of “tampered” notification, please follow the steps below:

1. Send a signing message which pretends to be a connection request by hiding the actual payload with new line characters.



2. Compare the previous screen to a real connection notification.



Impact

Potentially High. An attacker can potentially trick a user to sign malicious requests.

Complexity

High. The malicious sign notification will still have two visible differentiations from the valid connection request:

- "Sign" button instead of "Approve"
- "Sign Message" label instead of the "Connection" label

Considering the impact and complexity, the severity of this issue has been marked as “Low” only.

Remediation

Do not render special characters in the notification message. Additionally, add scrollbars to display the presence of larger content.

Appendix A - Vulnerability Classification

Vulnerability Severity	**Critical**
	High
	Medium
	Low
	Informational
Vulnerability Class	Components With Known Vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References (IDOR)
	Insufficient Authentication and Session Management
	Insufficient Authorization
	Insufficient Cryptography
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Condition
	Security Misconfiguration
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	User Privacy
	Time-of-Check to Time-of-Use (TOCTOU)

Appendix B - Remediation Checklist

The table below can be used to keep track of your remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

☐	**Upgrade the aforementioned dependencies to the latest version**
☐	**Don't store API keys in the code**
☐	**Enforce stricter permissions for sensitive files**
☐	**Strictly validate JSON-RPC method names**
☐	**Add trailing slashes to the external whitelisted domains. Consider introducing stricter string matching**
☐	**Verify that the `isFrameExtension` function allows whitelisted extension only**
☐	**Do not render special characters in the notification messages. Consider adding scrollbars**

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if you'd like more information on our retesting process.

Appendix C - Hardening

The following section covers aspect of the application that can be improved upon.

Implement TLS Certificate Pinning.

Certificate Pinning is the process of associating a host with its expected X509 certificate, public key, or chain of trust. The goal of certificate pinning is to prevent applications from accepting an imposter's TLS certificate that is used to pose as a legitimate service provider during Man-In-The-Middle attacks.
The absence of certificate pinning increases the risk of successful Man-In-The-Middle attacks when the application is used while being connected to untrusted networks (e.g. hotels, coffee shops, etc.).

Leverage the Windows App Store to provide signed updates.

Despite its popularity, we would suggest moving away from `Electron-Builder` due to the lack of secure coding practices and responsiveness of the maintainer.

Change device tag generation function does not use a cryptographically secure function.

```
frame-audit/main/rpc/index.js:17
      function randomLetters (num) {
        return [...Array(num)].map(() => String.fromCharCode(65 +
      Math.floor(Math.random() * 26))).join('')
      }
```

Use Node's `crypto` module instead.

Don't allow requests without Origin header.

All requests without the `Origin` header specified will have the same granted permissions.

```
frame-audit/main/api/trusted.js:26
       module.exports = async origin => {
        if (!origin || origin === 'null') origin = 'Unknown'
        if (invalidOrigin(origin)) return false
        if (origin === 'frame-extension') return true
```

For example, requests from an iframe with a sandbox attribute will not send the Origin header. If a user enables permission for any such iframe, all other iframes can use such 'Unknown' permissions. There is no way a user can distinguish from which iframe a particular request is coming.

Disable "nodeIntegrationInWorker" in the webPreferences.

If enabled, `nodeIntegration` allows JavaScript to leverage Node.js primitives and modules. This could lead to full remote system compromise if you are rendering untrusted content.

```
"webPreferences": {
        "nodeIntegrationInWorker": false
```